A PyTorch Implementation of a Spiking Neural Network Model

Mihaela Dimovska

Abstract

In this project, we analyze and re-implement the Spiking Neural Network model for digit classification, published in the article "Spiking neural networks for handwritten digit recognition—Supervised learning and network optimization", Shruti R. Kulkarni, Bipin Rajendran (Neural Networks, 2018). In particular, the forward pass is analyzed in detail, including the pixel-to-spikes conversion method, the generation of input currents, and the generation of spikes in the output layer from the input currents from the hidden layer. Python methods that implement each of these steps are provided. Furthermore, a modified version of the forward step is discussed, which significantly reduces the time needed for a single image forward pass, while producing similar spike-trains for the hidden-layer neurons. Thus, the contributions of this project are the detailed analysis and explanation of the forward pass of the SNN model, the Python/PyTorch implementation of the forward step of the SNN model (the scripts are self-contained and the user is required to just have installed the latest PyTorch version), and a variation of the forward steps which is significantly faster than the original forward method.

I. Introduction

Spiking Neural Networks (SNNs) are artificial networks that try to model the brain behavior as observed in nature. Indeed, instead of using static neurons with non-linear activation functions, SNNs model neurons that emit sparse spikes [1], [2]. Furthermore, the spike-generation process of an output neuron is governed by the spikes of the input neurons, meaning there is time-dependency in the learning process, unlike other types of networks whose neuronal behaviour is related to other neurons only via static operators. Learning via emitting and processing sparse spike signals makes SNNs promise extremely power-efficient computation [3]. This promise has been demonstrated with the rise of recent neuromorphic architectures that implement SNN models in hardware [2], [4], [5]. Executing machine learning problems on such a hardware and utilizing SNN models has already shown considerably smaller energy consumption as compared to utilizing other types of neural networks that run on conventional von-Neumann hardware [2], [5].

This exciting progress propels forward the cutting-edge research related to Spiking Neural Networks models [6], [7], [8], [4], [9]. Inspired by these exciting research works, this project aims to explore the details of one such SNN model, published in [6]. The work in [6] provides an SNN model that classifies the MNIST images [10] via supervised learning. In this project, we describe the forward pass of that model in detail and we provide Python scripts for each of the steps. Furthermore, a modified version of the forward step is discussed, which significantly reduces the time needed for a single image forward pass, while producing similar spike-trains for the hidden-layer neurons. Thus, the contributions of this project are the detailed analysis and explanation of the forward pass of the SNN model, the Python/PyTorch implementation of the forward step of the SNN model (the scripts are self-contained and the user is required to just have installed the latest PyTorch version), and a variation of the forward steps which is significantly faster than the original forward method.

The project report is organized as follows: section II gives an overview of the architecture/model of the network; section III describes the method that converts every pixel of an input image to a spike train; the subsequent sections detail the forward-pass mechanisms of each layer.

II. Model Overview

The SNN model consists of three layers of spiking neurons. As every image of the MNIST dataset has dimensions $28x28 = 784$, each input pixel is regarded as a spiking neuron, thus the network has 784 input spiking neurons. As a spiking neuron produces a spike-train, each pixel needs to be converted to spike-train. The process that converts a pixel value to a spike-train of length 1000 is described in the next section. Those spike trains are spatially convolved with 12 filters of dimensions $3x3$, over a period $T = 1000$. Namely, at every time-step $t, t = 1, ..., 1000$, the 784 spikes can be regarded as a $28x28$ input map, which can be convolved with the 12 filters, resulting in an output map of size $12x26x26$. As for every image this is happening for every time-step, we can regard every image as a "batch" of 1000 input maps, which will result in $1000x12x26x26$ output feature map after the convolution. Then, reshaping the output feature map, we in fact get $12x26x26 = 8112$ hidden-layer spiking neurons that will produce spike-trains of length 1000. The output map of the convolutional filters, actually produces current that is input to those 8112 hidden-layer spiking neurons. From that input current, the hidden layer neurons generate spikes; these spikes generate current that serves as input to the 10 output spiking neurons; from the input current the output neurons generate spikes. To summarize, the mechanisms of the model are the following:

- Every pixel value is converted to constant current for the input spiking neurons (Section III)
- From constant current, input spiking neurons generate spikes (Section III)
- From spikes of input neurons, current that is input to hidden layer neurons is generated via convolutional filters (Section IV)
- From current, hidden layer spiking neurons generate spikes (Section V)
- From spikes of hidden layer neurons, current that is input to output layer neurons is generated (Section VI)
- From current, output layer spiking neurons generate spikes (Section VII)
- Classification is done based on spike trains of the output neurons (Section VII)

Each of these steps is implemented in Python and detailed in the next sections.

This analysis of the forward mechanisms and the network architecture is summarized and depicted in Figure 1.

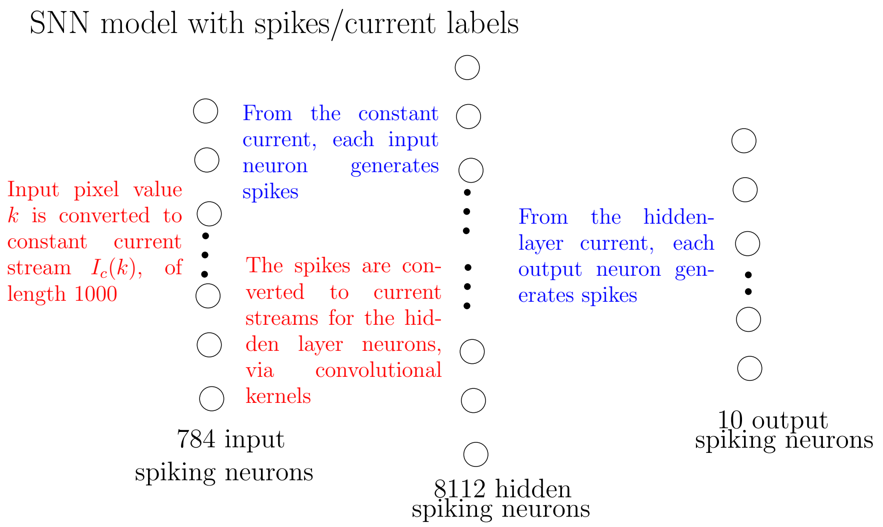
III. Pixel to Spike-Trains

The idea of converting pixels to spikes is that the bigger the value of the pixel, the more spikes the pixel should have in its spike train. As in the SNN, a neuron generates spikes as the result of applying current to that neuron, every pixel is first converted to a constant current input for the input neurons. Namely, for a pixel value k, the constant input current is a time-series of length 1000 with values: $I_k(t) = I_0 + (kI_p)$, for $t = 1, ..., 1000$, where $I_0 = 2700pA$ and $Ip = 101.2pA$. As k denotes the pixel value, $k = 0, ..., 255$.

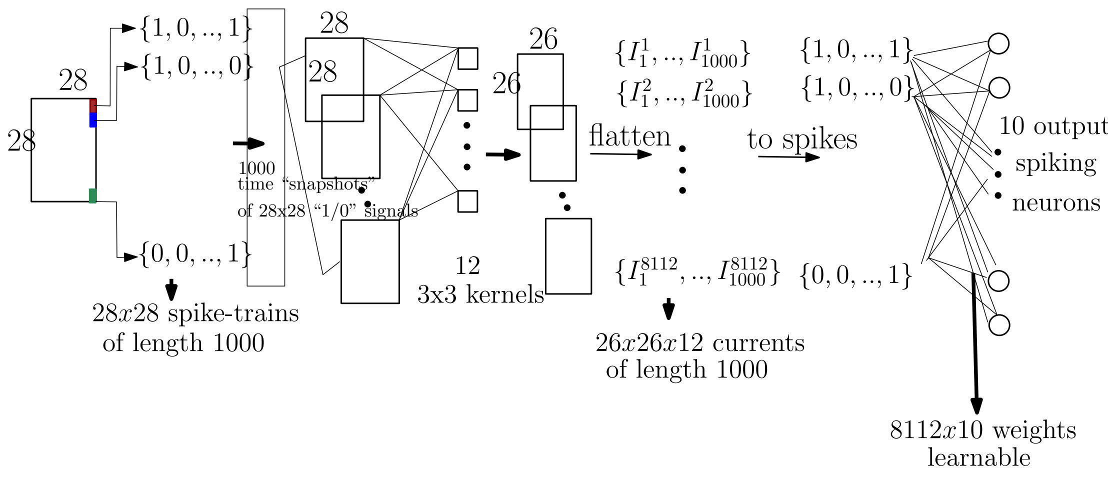
From the current input, the membrane potential of a neuron $V(t)$, grows according to the following equation:

$$C\frac{dV(t)}{dt} = g_L \cdot (V(t) - EL) + I_{syn}(t)$$

where $C = 300pF$ and $g_L = 30ns$ model the membrane's capacitance and leak conductance, respectively; $EL = 70mV$ is the resting potential; I_{syn} is the synaptic current. The threshold for firing is $VT = 20mV$ (i.e. if $V(t) \geq VT$ emit a spike) Biological neurons enter a refractory period immediately after a spike is issued during which another spike cannot be issued. This is implemented by holding the membrane potential at $V(t) = EL$ for a short refractory period $tref = 3ms$ after the issue


SNN model with spikes/current labels
Input pixel value k is converted to constant current stream $I_c(k)$, of length 1000
From the constant current, each input neuron generates spikes
The spikes are converted to current streams for the hidden layer neurons, via convolutional kernels
From the hidden-layer current, each output neuron generates spikes
784 input spiking neurons
8112 hidden spiking neurons
10 output spiking neurons

(a)


28
28
$\{1,0,..,1\}$
$\{1,0,..,0\}$
$\{0,0,..,1\}$
28x28 spike-trains of length 1000
1000 time "snapshots" of 28x28 "1/0" signals
28
28
12 3x3 kernels
26
26
flatten
$\{I_1^1,..,I_{1000}^1\}$
$\{I_1^2,..,I_{1000}^2\}$
$\{I_1^{8112},..,I_{1000}^{8112}\}$
26x26x12 currents of length 1000
to spikes
$\{1,0,..,1\}$
$\{1,0,..,0\}$
$\{0,0,..,1\}$
10 output spiking neurons
8112x10 weights learnable

(b)

Fig. 1. (a) The SNN model, via its 3-layer spiking neurons architecture, with a high-level explanation of the current/spikes generations in the layers. (b) More detailed representation of the mechanisms in the SNN model, with the role of the convolutional filters.

of a spike.

Having the input current I_{in}, we can calculate the membrane potential for each neuron at each time-step by, for example, using the Forward-Euler method:

$$V[:t+1] = V[:,t] + dt \cdot ((\frac{1}{capacity}) * (I_{in} - (gL \cdot (V[:,i] - EL)))) .$$

The initial state, $V[:,0] = EL$.

The method for generating spikes from input current is implemented in the script **from_current_to_spikes.py**, which takes as input the current time-series for every neuron.

In case of the input layer, the current is constant across time, for every neuron. Thus, every input neuron "issues spikes that are uniformly spaced in time, with a frequency that is sub-linearly proportional to the magnitude of its input current." Indeed, after generating the spike trains for every pixel $k = 0, ..., 255$, we plot the total number of spikes for each pixel in Figure 2 and we notice that the higher the value of the pixel, the more spikes its corresponding neuron has generated. The process of generating spikes from pixels is also detailed in the Jupyter notebook SNNProject.ipynb.

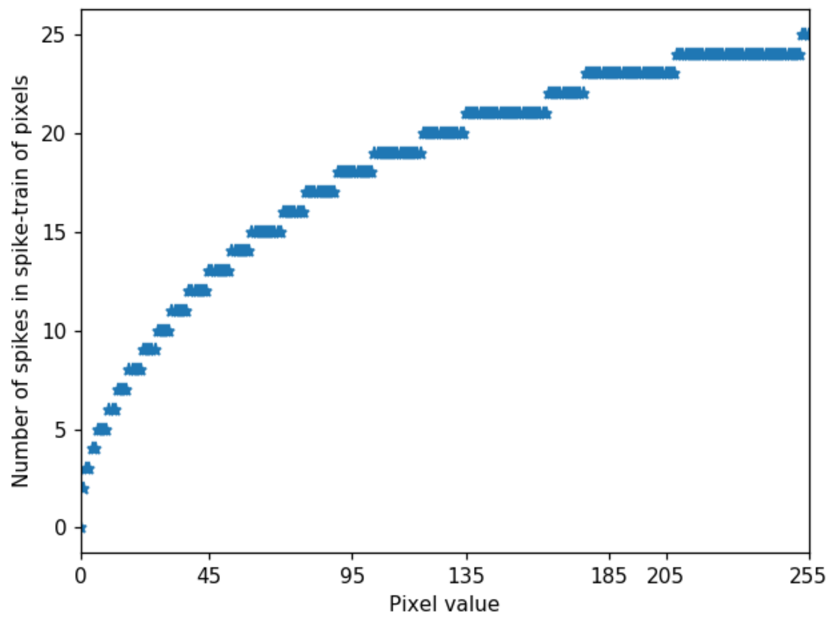


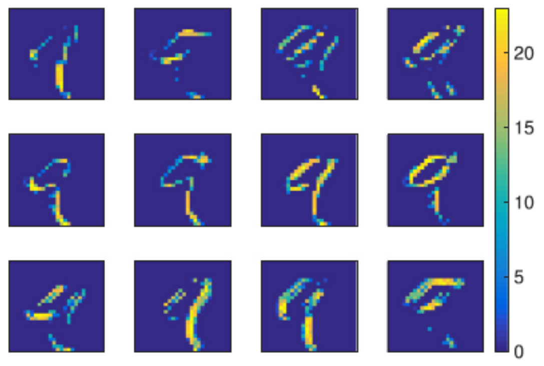
Fig. 2. Total number of spikes in the spike-trains of every pixel (each spike-train has length 1000.

IV. Input layer Spike-Trains to Currents for the Hidden Layer

Now that we know how to generate the spike trains of the input neurons, based on the value of the pixel that each input neuron receives (which in turn generates constant current that the input neuron receives), we need to generate input current for the hidden neurons from those spikes. As briefly explained in the Section II (Model Overview), at every time-step $t, t = 1, \ldots, 1000$, the spike trains from the 784 input neurons can be regarded as a $28x28$ input map, which can be convolved with the 12 filters. This results in an output map of size $12x26x26$. As for every image this is happening for every time-step, we can regard every image as a "batch" of 1000 input maps, which will result in $1000x12x26x26$ output feature map (tensor) after the convolution. Then, reshaping the output feature map, we in fact get $12x26x26 = 8112$ input currents, of length 1000, that feed-in to the 8112 hidden-layer spiking neurons. A visual representation of this process is provided in Figure 1(b).

V. Spike Generation in Hidden Layer

Taking the $8112x1000$ input current, and using the method for converting current to spikes (implemented in **from_current_to_spikes.py**) we obtain the spike-trains of the 8112 hidden-layer spiking neurons. As the spike matrix is of size $8112x1000$, we can re-shape it to $12x26x26x1000$, reflecting the spike generation based on each kernel. Then, summing up across the last dimension, we can visualize the number of spikes that each kernel produced. This is shown in Figure 3(b), and in Figure 3(a) the same visualization, as depicted in the article [6] is shown. Thus, we can see that we have reproduced this results successfully. The visualization is implemented in the script **sample_forward_pass_till_hidden_layer.py** Note that in this re-implementation, the convolutional filters are taken as the filters obtained after one epoch of training a neural network with just 1 convolutional layer (namely, consisting of 12 kernels of size $3x3$), without any bias.


(a)

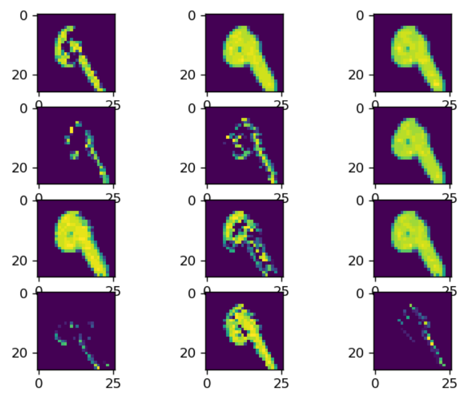

(b)

Fig. 3. Heat-maps of the total number of spikes, visualized when we sum across the last dimension of the $12x26x26x1000$ tensor (we visualize the 12 maps with size $26x26$ in terms of spike-count). (a) Such visualization in the article [6]; (b) Such visualization re-produced by the implementation in this project.

VI. From Hidden-Layer Spikes to Currents for the Output Layer

Once we have obtained the spike trains from the hidden layer, the next step is from those spikes to generate current that will be input to the output neurons. Based on that current, the output neurons will generate spikes.

The current in the hidden layer is calculated with the following method. First, from the spike-trains of the hidden neurons, synaptic kernels for every hidden neuron are calculated with $c(t) = \sum_i \delta(t - t^i) * (e^{-t/\tau_1} - e^{t/\tau_2})$ with $\tau_1 = 5ms$, $\tau_2 = 1.25ms$. The equation above, written more clearly, actually does the following at every time-step t when there is a spike: $c(t) = (e^{-1/\tau_1} e^{1/\tau_2}) * c(t-1) + 1$. Then, the current is calculated by $I_{syn}(t) = w \cdot c(t)$, where w is a matrix of $10x8112$ weights that are to be learned. Thus, the current that is input to the output spiking neurons is of shape $10x1000$, i.e. there is in-going current to all the 10 output spiking neurons, and that current is applied for 1000 time-steps.

The script **from_hidden_spikes_to_current_for_outputs.py** implements a function that does exactly what the script title says: converts the spikes-trains of the 8112 hidden spiking neurons to input current for the output neurons by calculating the synaptic kernels and then multiplying those kernels by the weights that are taken as input by the function.

VII. Spike Generation in Output Layer

The spikes in the output layer are generated are a result of the current coming from the hidden layer neurons, that is applied to the 10 output spiking neurons. Thus, just by using the method for current-to-spikes conversion (in the script **from_current_to_spikes.py**), with the current from the hidden-layer neurons as input to the method, we obtain the spikes in the hidden layer.

From the spike trains of the 10 output neurons, the neuron which had the most number of spikes is the one according to which classification is determined. Namely, every neuron $0, \ldots, 9$ corresponds to a digit and the input digit is classified as $k, k = 0, \ldots, 9$ is the k-th output neuron has the most spikes.

Finally, a full forward-step pass on one sample image is implemented in the script $forward\_step\_one\_sample.py$ the full forward-step with classification outcome is presented. Here is the full sequence of steps that are implemented in that script:

1) From pixels (input neurons) to spike trains of those input neurons

2) From the spike trains of the input neurons to current for the 8112 hidden neurons (using conv. filters)
3) From the input current to the hidden neurons, get the spike trains of the hidden neurons
4) The spike trains of the hidden neurons generate current to the 10 output neurons (here we use the weights and the synaptic kernels!)
5) From the input current to the output neurons, get the spike trains of the output neurons
6) Based on which output neuron has the highest number of spikes, classify

VIII. Modified Input layer Spike-Trains to Currents for the Hidden Layer

The spatial convolution of the input spikes with the convolutional kernels essentially converts every image to an input map with 1000 channels. Thus, in this network, it takes 1000 times longer for an input image to pass the first layer, as opposed to the same image being forwarded to the first layer (consisting of the same 12 filters) of purely convolution network. Indeed, the forward pass of just one image to the SNN model takes around 5 seconds on a GPU!

Thus, we propose a modification for faster evaluation. Namely, we propose the first layer to be fully convolutional. Then, similarly to the conversion of the pixels to spikes, we convert every value in the $12x26x26$ feature map as input current to the corresponding hidden-layer neuron. However, instead of regarding the values of the feature map as constant input currents, at every time t, we multiply the constant value by a corresponding decay kernel. Namely, for a value k in the output feature map (corresponding to the k-th neuron from the 8112 hidden layer neurons), we calculate its input current time-series as:

$$I_k(t) = k * e^{-t/\tau_1} .$$

Taking this current as input to the hidden layer neurons, we use the method for converting current to spikes to obtain the spike-trains of the hidden layer neurons. To validate this approach, we compare the average number of spikes (when a randomly chosen digit image is presented as input to the network) of the hidden neurons with this approach, versus the average number of spikes in the hidden neurons obtained with the approach described in [6]. A sample of the results with a randomly chosen sample digit of class 9 is forwarded as input is shown in Figure 4. We can see that though the second approach yields to slightly smaller average number of spikes, overall distributions are similar. Furthermore, as shown in Figure 4(c), visualizing the total number of spikes "per kernel" leads to similar results as the visualizations presented in [6]. The modification is presented in the script **sample_forward_pass_till_hidden_layer_mod.py**

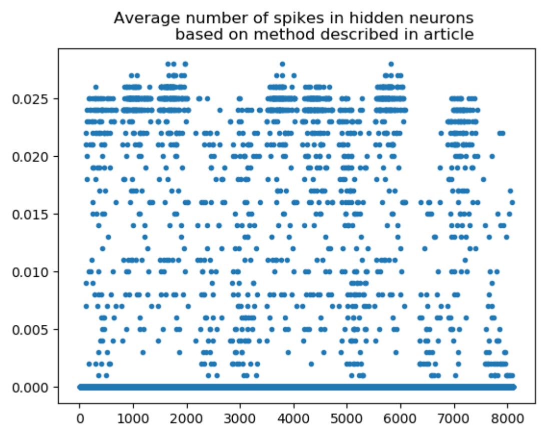
IX. Conclusion

In this project we studied an SNN model in detail and we implemented the forward pass from the training process of the SNN model, as described in the paper [6]. We reproduced several results from the paper, such as the frequency of spikes of the input and hidden-layer neurons. We also proposed using the convolutional layer entirely and converting each value in the output feature map of the convolutional layer to spikes. This makes the training process more efficient while achieving similar spiking frequency for the hidden layer neurons.

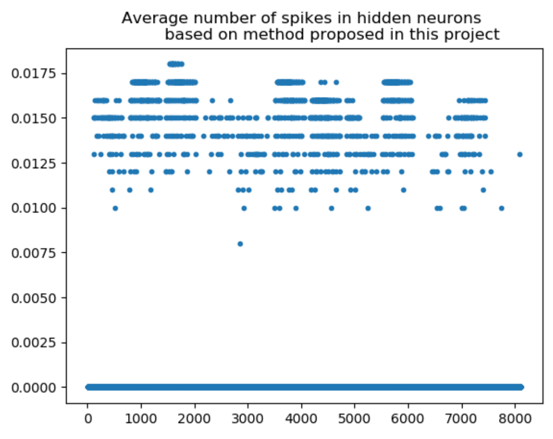
References

[1] H. Jang, O. Simeone, B. Gardner, and A. Grüning, "An introduction to spiking neural networks: Probabilistic models, learning rules, and applications [supplementary material]."

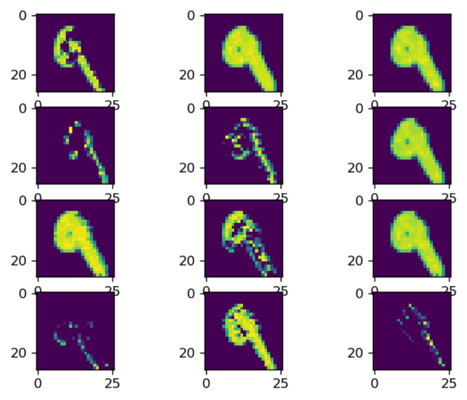
[2] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.



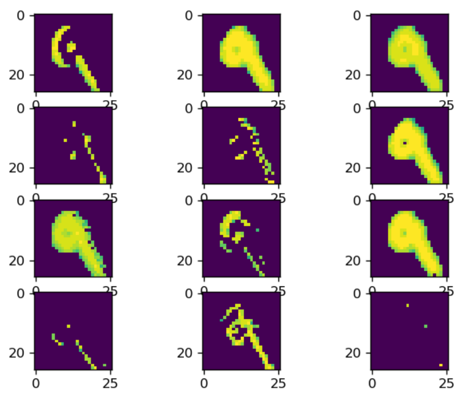
(a)



(b)



(c)



(d)

Fig. 4. (a) Average number of spikes in the spike-trains of the hidden neurons when those trains are obtained with the method in [6]; (b) Average number of spikes in the spike-trains of the hidden neurons when those trains are obtained with the modification described in this article. (c) Visualization of the frequency of the "kernel induced spikes" with the method in [6]. (d) Visualization of the frequency of the "kernel induced spikes" with the method described in this project.

[3] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv preprint arXiv:1705.06963*, 2017.

[4] J. S. Plank, C. D. Schuman, G. Bruer, M. E. Dean, and G. S. Rose, "The tennlab exploratory neuromorphic computing framework," *IEEE Letters of the Computer Society*, vol. 1, no. 2, pp. 17–20, 2018.

[5] J. Hsu, "Ibm's new brain [news]," *IEEE spectrum*, vol. 51, no. 10, pp. 17–19, 2014.

[6] S. R. Kulkarni and B. Rajendran, "Spiking neural networks for handwritten digit recognition—supervised learning and network optimization," *Neural Networks*, vol. 103, pp. 118–127, 2018.

[7] A. Bagheri, O. Simeone, and B. Rajendran, "Training probabilistic spiking neural networks with first-to-spike decoding," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 2986–2990.

[8] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, vol. 10, p. 508, 2016.

[9] C. S. e. a. Dimovska, Mihaela, "Multi-objective optimization for size and resilience of spiking neural networks," in *IEEE UEMCON*, 2019.

[10] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.